NPS-AM-08-031

EXCERPT FROM THE

PROCEEDINGS

OF THE

FIFTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

WHICH UNCHANGED COMPONENTS TO RETEST AFTER A TECHNOLOGY UPGRADE

Published: 23 April 2008

by

Dr. Valdis Berzins

5th Annual Acquisition Research Symposium of the Naval Postgraduate School:

Acquisition Research: Creating Synergy for Informed Change

May 14-15, 2008

Prepared for: Naval Postgraduate School, Monterey, California 93943

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
23 APR 2008		**00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Which Unchanged Components to Retest After a Technology Upgrade	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Computer Science Department,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
5th Annual Acquisition Research Symposium: Creating Synergy for Informed Change, May 14-15, 2008 in Monterey, CA

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**35**	

The research presented at the symposium was supported by the Acquisition Chair of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request Defense Acquisition Research or to become a research sponsor, please contact:

NPS Acquisition Research Program
Attn: James B. Greene, RADM, USN, (Ret)
Acquisition Chair
Graduate School of Business and Public Policy
Naval Postgraduate School
555 Dyer Road, Room 332
Monterey, CA 93943-5103
Tel: (831) 656-2092
Fax: (831) 656-2253
E-mail: jbgreene@nps.edu

Copies of the Acquisition Sponsored Research Reports may be printed from our website www.acquisitionresearch.org

Conference Website:
www.researchsymposium.org

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Proceedings of the Annual Acquisition Research Program

The following article is taken as an excerpt from the proceedings of the annual Acquisition Research Program. This annual event showcases the research projects funded through the Acquisition Research Program at the Graduate School of Business and Public Policy at the Naval Postgraduate School. Featuring keynote speakers, plenary panels, multiple panel sessions, a student research poster show and social events, the Annual Acquisition Research Symposium offers a candid environment where high-ranking Department of Defense (DoD) officials, industry officials, accomplished faculty and military students are encouraged to collaborate on finding applicable solutions to the challenges facing acquisition policies and processes within the DoD today. By jointly and publicly questioning the norms of industry and academia, the resulting research benefits from myriad perspectives and collaborations which can identify better solutions and practices in acquisition, contract, financial, logistics and program management.

For further information regarding the Acquisition Research Program, electronic copies of additional research, or to learn more about becoming a sponsor, please visit our program website at:

www.acquistionresearch.org

For further information on or to register for the next Acquisition Research Symposium during the third week of May, please visit our conference website at:

www.researchsymposium.org

THIS PAGE INTENTIONALLY LEFT BLANK

Which Unchanged Components to Retest after a Technology Upgrade

Presenter: Valdis Berzins is a Professor of Computer Science at the Naval Postgraduate School. His research interests include software engineering, software architecture, computer-aided design, and theoretical foundations of software maintenance. His work includes papers on software testing, software merging, specification languages, and engineering databases. He received BS, MS, EE, and PhD degrees from MIT and has been on the faculty at the University of Texas and the University of Minnesota. He has developed several specification languages, software tools for computer-aided software design, and fundamental theory of software merging.

Valdis Berzins
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
E-mail: berzins@nps.edu

Abstract

The Navy's open architecture framework is intended to promote reuse and reduce costs. This paper focuses on exploiting open architecture principles to reduce testing effort and costs in cases in which the requirements and code for a subsystem have not been changed, but the code is running on new hardware and/or new operating systems due to a technology-advancement upgrade. This situation is common in Navy and DoD contexts such as submarine, aircraft carrier, and airframe systems, and accounts for a substantial fraction of the testing effort. Unmodified software components need to be retested after a technology upgrade in some, but not necessarily in all cases. This paper reports some early research on conditions under which testing of unmodified components can be avoided after a technology upgrade, outlines an approach for identifying situations in which retesting can be safely reduced, and indicates how to focus retesting in cases in which it cannot be avoided.

Keywords: open architecture, reducing regression testing, automated testing, statistical testing, dependency analysis, reuse, operating system upgrades, hardware upgrades.

1. Introduction

The Navy is implementing the open architecture framework for developing joint interoperable systems that adapt and exploit open system design principles and architectures. Research being performed at the Naval Postgraduate School is pursuing a complementary effort to identify weaknesses and gaps in the current state of knowledge with respect to the development and testing of DoD/DoN systems according to such open systems principles, and to develop or adapt new methods for overcoming those weaknesses. The purpose of this effort is to provide sound engineering approaches to better realize the potential benefits of Navy open architectures and to provide concrete means that support economical acquisition and effective sustainment of such systems.

This project focuses primarily on improving test and evaluation of systems with open architectures, since this aspect can greatly benefit from improvements. Specific goals of this research are to enable the following: (i) reduction of unnecessary testing on every system

change, (ii) identification of what specific testing and checking procedures need to be repeated after changes, (iii) limiting the scope of retesting when the latter is necessary, and (iv) enabling a single analysis to provide assurance that all possible configurations that can be generated in a model-driven architecture will satisfy given dependability requirements. This paper reports some preliminary results of this project that address the first three of the goals listed above. A roadmap and technical approach for reaching the fourth goal are outlined in Berzins, Rodriquez and Wessman (2007).

The roadmap provides a long-term plan for eventually eliminating the need for regression testing after each reconfiguration and eventually enabling a "plug-and-fight" capability. This plan depends on the design and certification of a common architecture for a family of systems (FOS) that span a parameterized range of expected requirements, based on detailed standards for the components and connections. In this approach, the architecture is certified to meet its requirements, components are tested against standards and requirement parameters, and reconfiguration is achieved by swapping plug-compatible components with different requirement parameters (2007).

This paper focuses on the shorter-term problem of safely reducing testing for software components whose code has not been changed, without waiting for the results of long-term research and without relying on architecture-level certification.

The motivating context for the work reported here was to increase the effectiveness of quality assurance for Navy technology upgrades. The first step was to investigate conditions under which it is safe to reduce testing for software components whose code has not been changed, so that a larger fraction of the available time and effort could be focused on testing the new functionality introduced by the upgrade.

This focus was adopted after the author interviewed representatives from four of the organizations actually involved in developing such technology upgrades. These interviews indicated (with unanimous support) that those organizations' highest current priorities are reducing testing for unmodified software components after a technology upgrade and adapting automated testing methods into production use. The initial research, therefore, explored practical methods for checking conditions under which it is safe to reduce or eliminate retesting for unchanged components, and sought solutions that leverage automated testing in the contexts in which it is easiest and most effective to do so.

Technology upgrades are typically performed on a two-year cycle. They often involve migration to the best hardware and operating system version available at the time, where "best" implies a balanced tradeoff between high performance and reliable operation. Typically, only a small fraction of the application code has been changed. However, current certification practices require all of the code to be retested prior to deployment, whether it has been modified or not. Retesting of an unchanged module can be avoided only if we can establish that it has not been adversely impacted by the change. The rest of this paper explores ways to determine that, and the conditions under which such determination is possible.

The rest of this paper is organized as follows. Section 2 describes methods for deciding when re-testing of unchanged components can be safely reduced or eliminated entirely. Section 3 discusses the costs of the automated testing of operating system services needed to support some of the methods presented in Section 2. Section 4 explains the significance of *operational profiles* (probability distributions characterizing expected workloads for software services), which are also needed to support the types of automated testing needed by methods proposed in

Section 2. Section 5 identifies the conditions under which unchanged code does need to be tested, along with the potential failure modes that may need to be guarded against and how to focus the retesting to guard against these modes without repeating previous testing effort. Section 6 identifies some relevant previous work, and Section 7 concludes with a summary of the steps that should be taken to enable practical application of the test-reduction approach presented in this paper.

2. Deciding When Retesting Can Be Avoided

If the requirements related to a component have not changed, and the behavior of the components has not changed, then retesting may not be necessary. As discussed further in Section 4 of this paper, the range of conditions under which a component is expected to provide its operational capabilities is a part of its requirements that is particularly relevant to testing and re-testing. The rest of this section addresses how to statically and dynamically check that the behavior of a component has not changed, assuming for the moment that its requirements and range of operating conditions have not changed.

A type of dependency analysis known as program slicing can be used to identify parts of the unchanged code that have the same behavior in the new release as in previous one (Weiser, 1984, July). A program slice at a given observation point is a self-contained subset of the code in the sense that it contains all of the code that can affect the behavior visible at the observation point. If two different programs have the same slice for a given observation point, then they have the same visible behavior at that point. Consequently, if the new release has the same slice as the old release for a given service, then that service will have exactly the same behavior in the new release as in the old one and, consequently, may not need regression testing (Gallagher, 1991, August). This fact is useful because program slices can be computed for software systems on practical (large) scales. The testing-reduction method that follows from this observation is to compute the slice of each service with respect to the new release and the old release, and retest only the services for which these slices differ.

In the context of technology-advancement upgrades, the test-reduction method described above must be augmented with focused, automated testing to produce a substantial reduction in retesting. Technology upgrades usually run on a new version of the operating system. If the source code of the operating system is proprietary and, hence, not available for static analysis (commonly true, except for open source systems such as LINUX), then the only safe assumption is that all operating system services have been impacted by the upgrade to the new version. Thus, any service whose slice includes a dependency on a system call would be potentially impacted and would have to be retested, based on the simple slicing approach outlined in the previous paragraph. This is likely to include most of the application-level modules, thus severely limiting the amount of savings that can be obtained using slicing alone.

Automated testing, however, can enable larger reductions in retesting if it is focused on the middleware interface to the underlying operating system services. Fortunately, the author's interviews with representative stakeholders confirmed that most Navy systems with open architectures are designed around a middleware interface that encapsulates all operating system calls. Such middleware interfaces are also prevalent in other DoD systems, including the US Army's FCS. Application architectures are typically designed in this way to ease the job of porting the application to new operating systems, whether they are new releases of the same product or different products. Consequently, each new release of the operating system and the neighboring middleware layer are both designed to preserve the observable behavior of the previously available system calls if at all possible—even if the details of the implementation may

vary from one release to the next. If we know that the observable behavior of a given system call is the same in the old and the new version of the operating system, then we can truncate the slice at the middleware layer for that call, and conclude that the behavior of an application service is unaffected by the OS change if its abbreviated slices in the two versions are the same. The proposed enhancement to dependency analysis using program slicing is to check this property for each system call in the middleware layer via automated testing.

This same strategy can also be applied at higher levels of middleware. For example, for the common case of applications that have been developed for the Java or .NET platforms, the interface to operating system resources is the framework runtime, such as the interface to the Java foundation classes. One related viable strategy for reducing testing of unchanged application code is bounding slicing by the interfaces at this level and using automated testing to show equivalent behaviors of the two releases at these interfaces. A related, common pattern of changes that should not affect behavior involves framework evolution, in which applications are recoded to migrate from "deprecated" (soon to become obsolete) interfaces to the corresponding new versions of the interfaces. Although such changes produce differences in the code, they are intended to preserve behavior, and should be amenable to the automated test strategy. Thus, modules one level above the framework runtime interfaces are additional candidates for automated testing and slicing cutoff boundaries.

Automated testing is attractive in these contexts because a simple, reliable implementation of a "test oracle" is possible for the encapsulated operating systems services. A "test oracle" is a process for automatically determining which test outputs pass and which ones fail. The "unchanged behavior" condition can be easily checked by software for a given set of input data. This is possible since both the old and the new versions of the operating system are available for testing, and test scaffolding software can compare the results of the two versions via equality tests. The existence of such a "test oracle" implies that the OS middleware testing process can be completely automated—enabling economic and practical testing with statistically significant sample sizes that support very high confidence levels, or, in some cases, even exhaustive testing of the operating system interfaces that supports definite conclusions. The proposed automated testing process would, thus, classify all of the services in the middleware interface to the operating system into two groups: those whose behavior is the same in both versions of the operating system (the preserved services), and those whose behavior differs in the two versions (the modified services). We expect the first group to be much larger than the second group.

In such cases, we can cut off slices at the system calls to the preserved services, and conclude that unmodified application components do not have to be retested unless their slices differ or contain system calls that invoke one of the modified services. The operating system interface always needs to be thoroughly retested, but this can be done by the affordable automated process described above.

The above analysis depends on the assumption that we can accept a statistical inference about the unchanged behavior of the operating system's calls, if the statistical confidence level is high enough. Since most military decisions must be based on information that has the same degree of uncertainty, we do not expect lack of certainty to be a problem in principle. We, therefore, consider how to determine what level of confidence would be "high enough" and how many test cases are necessary to reach that level of confidence.

We start with a consideration that should be meaningful to the stakeholders: if the mean time between observations of a behavioral difference in a given operating systems service is

substantially (k times) longer than a mission, it is acceptable to ignore risks due to the possibility of such an unexpected difference. The meaning of "substantially" can be expressed as a numerical safety factor k that can be understood and set by system stakeholders based on their tolerance for risk.

Next, we measure the mean number of executions per mission es for each service s in the middleware interface to the operating system. The objective of the automated testing for each service s is to ensure the mean number of executions between observed differences in the behavior of service s is at least Ns, where

$$N_s = k\, e_s.$$

Theorem 4.3 from Howden (1987) can then be used to determine the required number of test cases Ts for each service:

$$T_s = N_s \log_2 N_s.$$

If we run Ts test cases that are independently drawn from the probability distribution characterizing the mission (called the operational profile), the theorem will enable us to conclude that the mean number of executions is at least Ns with a statistical confidence level (1 – 1/Ns); however, this is contingent upon none of the Ts test cases showing any differences in the behavior of the services under the new version of the operating system from those in the previously released version.

The rationale for this choice of confidence level is that it makes the probability of making a false positive conclusion no more than the acceptable frequency of behavioral differences, thus scaling the risk due to random sampling errors to match the specified maximum acceptable failure rate. False positive conclusions correspond to cases in which the frequency of behavioral differences in the new release of the operating system service in question is actually greater than the target bound (1/Ns), but the automated testing procedure failed to observe a difference due to random sampling fluctuations that caused conforming results to appear purely by chance. The test set size Ts has been chosen to make the probability of such a chance observation at most (1/Ns).

Thorough statistical testing of the operating system interfaces has the additional benefits of increasing confidence that differences in hardware (and possibly different versions of the compilers, linkers and loaders) have not affected the behavior of the applications built using these services.

3. Cost of Automated Testing

There are several different kinds of automated testing. The most common kind is semi-automated testing. This approach automates the type of testing currently performed manually. It is commonly the first kind of automated testing implemented in an organization because it does not involve any process changes. In this type of approach, the test cases are still developed individually by test engineers, but the test cases are run automatically, and the results are classified into pass or fail categories automatically—often by comparison to previously captured test outputs that were originally individually examined and categorized by people. In this approach, execution and categorization of test results is automated, but the choice of test cases and the initial pass/fail decisions are not. This approach saves appreciable time and effort

relative to a completely manual approach, but the human effort required is still proportional to the number of test cases.

Another approach particularly relevant in our context is automated statistical testing. In this approach, the choice of test cases and the initial pass/fail decisions are automated, as well. This makes a great difference because the human effort involved does not increase with the number of test cases to be executed. This enables economical application of the very large test sets needed to achieve the coverage required to support high levels of statistical confidence in the dependability of the software. The high levels of statistical confidence are needed to avoid testing for other unchanged code based on indirect evidence that the behavior of the underlying services on which the unchanged code depends has not changed.

The context identified in the previous section is well suited for automated statistical testing, because the choice of test cases and the initial pass/fail decisions are easily automated in that context: the first can be done by random sampling from the operational profile, and the second by comparison of the results produced by the previous release of the software to those produced by the new release.

The variation in the number of the test cases Ts required as a function of the acceptable risk of false positive conclusions (1/Ns) is illustrated in Table 1.

Table 1. Number of Test Cases Required for Different Levels of Risk Tolerance

N_s	C	T_s
10^3	.999	1.0×10^4
10^4	.9999	1.3×10^5
10^5	.99999	1.7×10^6
10^6	.999999	2.0×10^7
10^7	.9999999	2.3×10^8
10^8	.99999999	2.7×10^9
10^9	.999999999	3.0×10^{10}

Ns: Desired lower bound on mean number of executions between differences
C: Statistical confidence level
Ts: Number of independent random test cases required

Figure 1 shows how the cost characteristics of the proposed automated testing approach compare to the costs of manual testing. The cost curves are close to straight lines; the fixed costs of automated testing are larger than for manual testing, and the marginal cost of adding another test case is much smaller for automated testing than for manual testing.

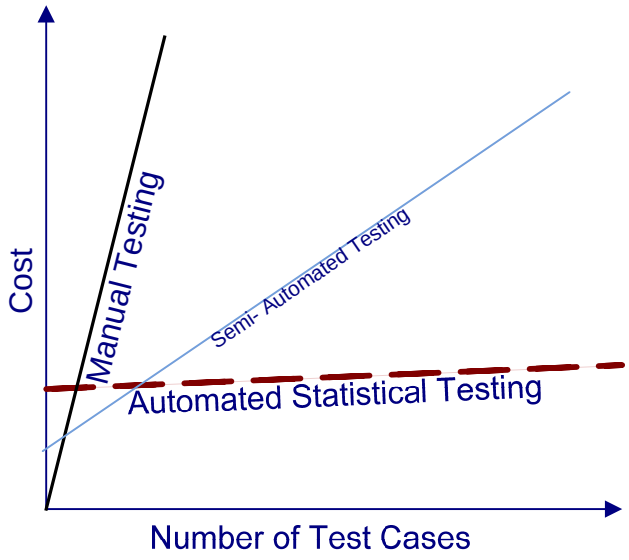


Figure 1. Testing Cost Characteristics

In order to determine the crossover points, we must have experimental data. However, we expect automated testing to be affordable—even for the very large numbers of test cases needed for high confidence in stability of OS services across different releases. We also expect manual and semi-automated approaches not to be affordable when we test to high confidence.

Regarding the time and other resources to perform the proposed automated statistical testing, we can note the following:

1. It typically takes a small amount of time to perform a single system call.
2. Testing using independent, random samples is easily parallelizable and could be effectively spread over large numbers of processors using well-established techniques—such as Google's Map Reduce programming model (Lammel, 2008, January)—if very high confidence levels are needed.
3. Behavior of operating system calls can be tested independently of other shipboard systems and does not require interactions with human operators.

Since the testing process is completely automated, the variable cost of these tests is due to computing time and hardware, but not to human effort. The benefit of the automated statistical test approach described here is that there are no variable costs for labor. Since computing resources are currently inexpensive and steadily getting cheaper, this implies that even the relatively large numbers of test cases needed for high confidence are likely to be affordable.

This approach does involve some fixed costs for human effort that may be higher than in less-disciplined manual approaches. These costs are due to the need for the following activities:

1. Measurement of operational profiles—i.e., the frequency distributions of operating system calls and their associated input parameters. Instrumented versions of the software can be used during exercises to collect measurements of the operational

profiles, or, if the computational overhead of doing this is acceptable, measurements could also be collected during actual operations.

2. Coding more sophisticated test-driver software that includes code for generating random samples from the measured operational profiles, code that implements test oracles as described in Section 2, and code that keeps track of testing statistics and reports them.

4. Why Do We Care about Operational Profiles?

Accurate estimates of operational profiles, preferably based on actual measurements, are necessary because in all practical cases, the reliability of a software system is meaningless without firm knowledge of the operational profile. This claim is based on the hypothesis that all real systems have at least one input value x for which they perform correctly, and at least one other input value y for which they do not. If we know x and y, we can construct a spectrum of possible operational profiles for which the reliability of the same system ranges from 0 to 1 and attains every value in between.

The above line of reasoning shows that the only systems whose reliabilities do not depend on the operational environment are those that fail for all possible inputs (reliability uniformly 0, not interesting), and those that operate correctly for all possible inputs (reliability uniformly 1, not attainable in practice for large systems).

For all other systems, the reliability is determined by the operational profile and can vary widely for different operational contexts. This has serious implications for component reuse, which is a cornerstone of the open Navy architecture initiative.

Operational profiles have been used by the testing research community for many years and have been applied in many contexts. For example, they have been measured and used to assess the reliability of telephone-switching software.

5. When Retesting Is Needed

If the process described in Section 2 shows that the slice of a given application level service differs in the new and previous release, then behavior of the system has been impacted, and the service needs retesting. Services whose requirements have changed will be in this category—so new functionality needs to be tested according to the criteria proposed in Section 2, as expected. If the behavior of unchanged modules with unchanged requirements can be affected by other modified modules, they will be also identified by the slicing process. These also need retesting to check if there are any unintended indirect consequences of the code changes. This is the effect most developers and test and evaluation organizations are concerned about guarding against.

In addition, however, some modules may need to be retested even if their behavior has not changed, because reliability of a system depends on its environment as well as on its implementation. Thus, changes to the environment of a system can affect its reliability even if the behavior of the system remains unchanged. This possibility must be considered in the context of Navy open architectures because they strongly encourage reuse of components in a variety of operational environments to provide cost savings.

When a reusable component is moved unchanged to a new operating environment, we need to check whether its range of expected operating conditions has expanded—as

manifested by an expanded range of expected input parameter values in its new operating environment in comparison to its previous operating environment. If this is the case, then the component needs to be tested both on samples from the previously untested part of the input space, as well as on scenarios typical of the novel features of the new operating environment for the reusable component. If the analysis of the operating environment is done properly, we will not have to repeat the tests conducted previously, but rather must run new and substantially different tests that focus on the new situations that are likely in the new operating environment but were not likely in the previous ones.

One other issue to be considered is whether the requirements of the component involve timing constraints. The above discussion has focused mostly on the functional behavior of the component, and not on how much time it takes to produce those results. Components that are subject to strict timing requirements need additional quality-assurance; analysts must check those requirements if the characteristics of the hardware in the new release will differ from those in the old one. Perhaps surprisingly, this is the case even if the new hardware is faster than the old hardware. This is due to the properties of the scheduling methods to be used. In particular, it is known that rate-monotonic scheduling, one popular method for scheduling real-time software, is (in some cases) subject to anomalies. For instance, a given schedule may work fine for a given hardware configuration but may miss deadlines when executed on faster hardware. This can happen if uninterruptible operations or those that lock shared resources are executed in a different order on the new hardware—due to the completion of a sped-up task prior to the release time of a competing task that was previously unreachable. Methods for checking dependencies on timing constraints are beyond the scope of this paper.

To focus retesting where it is needed most, the author recommends the establishment of an explicit process to track past and projected changes in operational profiles and to reflect these changes in testing plans. Some preliminary steps in this direction are to:

1. Keep records of operational profiles used in testing previous releases of subsystems.
2. Measure operational profiles under mission conditions and exercises exploring new concepts of operations. Check for differences from those covered in past testing.
3. Focus retesting efforts on circumstances and scenarios that have weight in actual and projected operational profiles that have not been covered well in previous testing of the same unchanged components.

6. Relevant Previous Work

Program slicing has been used in a wide variety of applications, including testing (Binkley, 1998; Gupta, Harrold & Soffa, 1992; Harman & Danicic, 1995; Hierons, Harman & Danicic, 1999; Hierons, Harman, Fox, Ouarbya & Daoudi, 2002), debugging (Agrawal, DeMillo & Spafford, 1993; Lyle & Weiser, 1987), program understanding (De Lucia, Fasolino & Munro, 1996; Harman, Hierons, Danicic, Howroyd & Fox, 2001), reverse engineering (Canfora, Cimitile & Munro, 1994), software maintenance (Gallagher, 1991, August; Cimitile, De Lucia & Munro, 1994), change merging (Horwitz, Prins & Reps, 1989; Berzins & Dampier, 1996), and software metrics (Lakhotia, 1993; Bieman & Ott, 1994). More detailed surveys of previous work on slicing can be found in Binkley and Harmon (2004). Although the subject is outside the scope of the current paper, which focuses on reducing testing by detecting unintended interactions between different parts of a program, Gallagher (1991, August) also outlines a method for preventing the introduction of new unintended interactions during software upgrades.

Automated testing has been studied in a wide variety of contexts. An approach to automatically generating test-driver code from formal requirements is described in Berzins and Chaki (2002). This approach automatically generates open sets of test cases based on random samplings from implementations of operational profile distributions. The pass/fail decisions that classify the results produced by the individual test cases are made by software methods that are automatically generated from the requirements, which must be sufficiently precise and constructive to support this process. The number of test samples in the generated test set is automatically set to meet specified reliability goals expressed in terms of mean number of executions between failures. This work provides an approach to extending automated statistical testing to contexts beyond those in which the expected behavior of a module is unchanged in the new release.

There has also been previous work on quality assurance for flexible systems at the levels of requirements (Luqi, Zhang, Berzins & Qiao, 2004, December; Luqi & Lange, 2006, November 8) and architectures (Berzins & Luqi, 2006, May 6; Luqi & Zhang, 2006, May 6). In addition, a method for assessing the impact of timing constraints on reliability of system upgrades can be found in Qiao, Wang, Luqi, and Berzins (2006, March).

7. Conclusion

Further research is recommended to substantiate the practical applicability of the ideas outlined above. Experimental evaluation of the slicing method for identifying modules that do not have to be rested should be performed, together with the focused automated testing methods needed to fully realize the potential savings of the approach.

Measurement and analysis of the operational profiles of reusable components can be used to support analysis of changes in the operating environment that may require focused retesting of components whose behavior has not changed. Operational profiles are probability distributions that serve as mathematical representations of the operating environment and that are needed to support statistically significant testing that can reduce the testing effort, as described above. These distributions can be measured by instrumenting components and collecting statistics as they run, either in exercises or during actual missions, and can be used to drive statistically based automated testing that can quantitatively assess the reliability of systems.

Although it is not easy to convince contractors to automate their testing if they are not familiar with this approach, the economic incentives to do so are getting more compelling. This practical problem is particularly evident in the current situation—in which domain experts are often doing the project management and coding with little knowledge of or experience with recent advances in the techniques and tools used in software engineering. The increasing popularity of agile methods, which depend heavily on semi-automated testing, should help change this perception. Pilot projects demonstrating the effectiveness of the suggested approach are recommended to provide concrete data about costs and benefits, thereby alleviating concerns about project risks due to technology innovations.

List of References

Agrawal, H., DeMillo, R., & Spafford, E. (1993). Debugging with dynamic slicing and backtracking. *Software Practice and Experience, 23*(6), 589–616.

Berzins, V., & Chaki, N. (2002, October 7-11). MTDOAG: Module Test Driver and Output Analyzer Generator. In *Proceedings of the 9th Monterey Workshop: Radical Innovations of Software and Systems Engineering in the Future* (pp. 48-56). Venice: Universita Ca Foscari di Venezia.

Berzins, V., & Dampier, D. (1996). Software merge: Combining changes to decompositions. *Journal of Systems Integration, 6*(1-2, special issue on Computer-aided Prototyping), 135-150.

Berzins, V., & Luqi. (2006, May 6). Achieving dependable flexibility via quantifiable system architectures. In *Proceedings of Workshop on Advances in Computer Science and Engineering* (pp. 53-54). Berkeley, CA .

Berzins, V., Rodriguez, M., & Wessman, M. (2007, May 16-17). Putting teeth into open architectures: Infrastructure for reducing the need or retesting. In *Proceedings of the Fourth Annual Research Symposium—Acquisition Research: Creating Synergy for Informed Change* (pp. 285-311). Monterey, CA: Naval Postgraduate School.

Bieman, J., & Ott, L. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering, 20*(8), 644–657.

Binkley, D. (1998). The application of program slicing to regression testing. In M. Harman & K. Gallagher (Eds.), *Program Slicing, Information and Software Technology, 40*(11-12), (pp. 583-594) (special issue).

Binkley, D.W., & Harman, M. (2004). A survey of empirical results on program slicing. In M.V. Zelkowitz (Ed.), *Advances in Computers* (pp. 105–178). (Vol. 62). San Diego, CA: Elsevier.

Canfora, G., Cimitile, A., & Munro, M. (1994). RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice, 6*(2), 53–72.

Cimitile, A., De Lucia, A., & Munro, M. (1996). A specification driven slicing process for identifying reusable functions. *Software Maintenance: Research and Practice, 8*(3), 145–178.

De Lucia, A., Fasolino, A., & Munro, M. (1996). Understanding function behaviours through program slicing. In *Proceedings*, *4th IEEE Workshop on Program Comprehension* (pp. 9–18). Los Alamitos, CA: IEEE Computer Society Press.

Gallagher, K. (1991, August). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering, 17*(8), 751-760.

Gupta, R., Harrold, M., & Soffa, M. (1992). An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance* (pp. 299–308). Los Alamitos, CA: IEEE Computer Society Press.

Harman, M., & Danicic, S. (1995). Using program slicing to simplify testing. *Software Testing, Verification and Reliability, 5*(3), 143–162.

Harman, M., Hierons, R., Danicic, S., Howroyd J., & Fox, C. (2001). Pre/post conditioned slicing. In *Proceedings*, *IEEE International Conference on Software Maintenance ICSM'01* (pp. 138–147). Los Alamitos, CA: IEEE Computer Society Press.

Hierons, R., Harman, M., & Danicic, S. (1999). Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability, 9*(4), 233–262.

Hierons, R., Harman, M., Fox, C., Ouarbya, L., & Daoudi, M. (2002). Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability, 12*(1), 23–28.

Horwitz, S., Prins, J., & Reps, T. (1989). Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems, 11*(3), 345–387.

Howden, W. (1987). *Functional program testing and analysis*. New York: McGraw-Hill.

Lakhotia, A. (1993). Rule-based approach to computing module cohesion. In *Proceedings of the 15th Conference on Software Engineering, ICSE-15* (pp. 34–44). Los Alamitos, CA: ACM/IEEE.

Lammel, R. (2008, January). Google's map reduce programming model—Revisited. *Science of Computer Programming*, *70*(1), 1-30.

Lyle, J., & Weiser, M. (1987). Automatic program bug location by program slicing. In *Proceedings*, *2nd International Conference on Computers and Applications* (pp. 877–882). Los Alamitos, CA: IEEE Computer Society Press.

Luqi, & Lange, D. (2006, November 8). Schema changes and historical information in conceptual models in support of adaptive systems. In *Proceedings*, *First International Workshop on Active Conceptual Modeling of Learning* (pp. 112-121). Tucson, AZ: Springer LNCS 4512.

Luqi, & Zhang, L. (2006, May 6). Documentation Driven Evolution of Complex Systems. In *Proceedings of Workshop on Advances in Computer Science and Engineering* (pp. 141-170). Berkeley, CA.

Luqi, Zhang, L., Berzins, V., Qiao, Y. (2004, December). Documentation driven development for complex real-time systems. *IEEE Transaction on Software Engineering, 30*(12), 936-952.

Qiao, Y., Wang, H., Luqi, & Berzins, V. (2006, March). An admission control method for dynamic software reconfiguration in complex embedded systems. *International Journal of Computers and Their Applications, 13*(1), 28-38.

Weiser, M. (1984, July). Program slicing. *IEEE Transactions of Software Engineering, SE-10*(4), 352-357.

2003 - 2008 Sponsored Research Topics

Acquisition Management

- Software Requirements for OA
- Managing Services Supply Chain
- Acquiring Combat Capability via Public-Private Partnerships (PPPs)
- Knowledge Value Added (KVA) + Real Options (RO) Applied to Shipyard Planning Processes
- Portfolio Optimization via KVA + RO
- MOSA Contracting Implications
- Strategy for Defense Acquisition Research
- Spiral Development
- BCA: Contractor vs. Organic Growth

Contract Management

- USAF IT Commodity Council
- Contractors in 21st Century Combat Zone
- Joint Contingency Contracting
- Navy Contract Writing Guide
- Commodity Sourcing Strategies
- Past Performance in Source Selection
- USMC Contingency Contracting
- Transforming DoD Contract Closeout
- Model for Optimizing Contingency Contracting Planning and Execution

Financial Management

- PPPs and Government Financing
- Energy Saving Contracts/DoD Mobile Assets
- Capital Budgeting for DoD
- Financing DoD Budget via PPPs
- ROI of Information Warfare Systems
- Acquisitions via leasing: MPS case
- Special Termination Liability in MDAPs

Human Resources

- Learning Management Systems
- Tuition Assistance
- Retention
- Indefinite Reenlistment
- Individual Augmentation

Logistics Management

- R-TOC Aegis Microwave Power Tubes
- Privatization-NOSL/NAWCI
- Army LOG MOD
- PBL (4)
- Contractors Supporting Military Operations
- RFID (4)
- Strategic Sourcing
- ASDS Product Support Analysis
- Analysis of LAV Depot Maintenance
- Diffusion/Variability on Vendor Performance Evaluation
- Optimizing CIWS Lifecycle Support (LCS)

Program Management

- Building Collaborative Capacity
- Knowledge, Responsibilities and Decision Rights in MDAPs
- KVA Applied to Aegis and SSDS
- Business Process Reengineering (BPR) for LCS Mission Module Acquisition
- Terminating Your Own Program
- Collaborative IT Tools Leveraging Competence

A complete listing and electronic copies of published research are available on our website: www.acquisitionresearch.org

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CALIFORNIA 93943

www.acquisitionresearch.org

Which Unchanged Components to Retest after a Technology Upgrade

Valdis Berzins
Professor – Department of Computer Science (NPS)
E-mail: berzins@nps.edu, Phone: 831-656-2610

Context

- The Navy is moving towards an Open Architecture paradigm
  - Joint interoperable systems that ***adapt*** and are built using open interfaces, open design principles, and open architectures
- Expected long term benefits from Navy Open Architecture
  - Business benefits:
    - Flexible acquisition strategies and contracts that enable ***software reuse, easy systems upgrade***, and ***shared data*** throughout the Navy
  - Technical benefits:
    - Modular open architectures facilitate ***portability***, maintainability, interoperability, ***upgrade-ability*** and ***long-term supportability***
- The Achilles Heel - Test and Evaluation
  - Current practices require ***retesting unchanged components*** in each new deployment context, typically every two years
  - Substantial budget and schedule are currently devoted to retesting
  - ***New technology, processes, and policies*** are needed to ***safely reduce*** this effort and free resources for testing new functionality

Objectives

- ***Safely reduce*** software system testing cost

- Software system testing cost consists of
  - Up-front testing cost

    PLUS
  - Cost attributed to missed errors
    - I.e., cost of future system failures

- We seek to reduce both parts of the cost

Problem Statement

- According to Navy and other experience, traditional approaches to testing are not well suited to open environments
  - They are ***too expensive***, ***take too long*** and ***lack agility*** to react to changes during acquisition or missions
  - Have to be ***repeated after every change***

- Typical testing assumptions are not valid for Open Architectures
  - Conventional testing methods require the ***system environment*** to be ***fixed*** and ***known in detail*** at test and evaluation time
    - Effectiveness of testing is very sensitive to the expected operating environment, which is ***unknown for reusable components***
  - Current test and evaluation methods check conformance to ***specifications***
    - The majority of ***failures*** in software systems are due to requirements and ***specification errors***, and commonly show up after a subsystem has been ***moved to a different environment***
    - Commonly called “system integration problems”

Approaches

- Reduce testing cost (this paper)
  - Methods to ***identify components that do not need to be retested***
  - Methods to ***limit scope of retesting*** when it is needed
  - Methods to ***completely automate*** testing and analysis

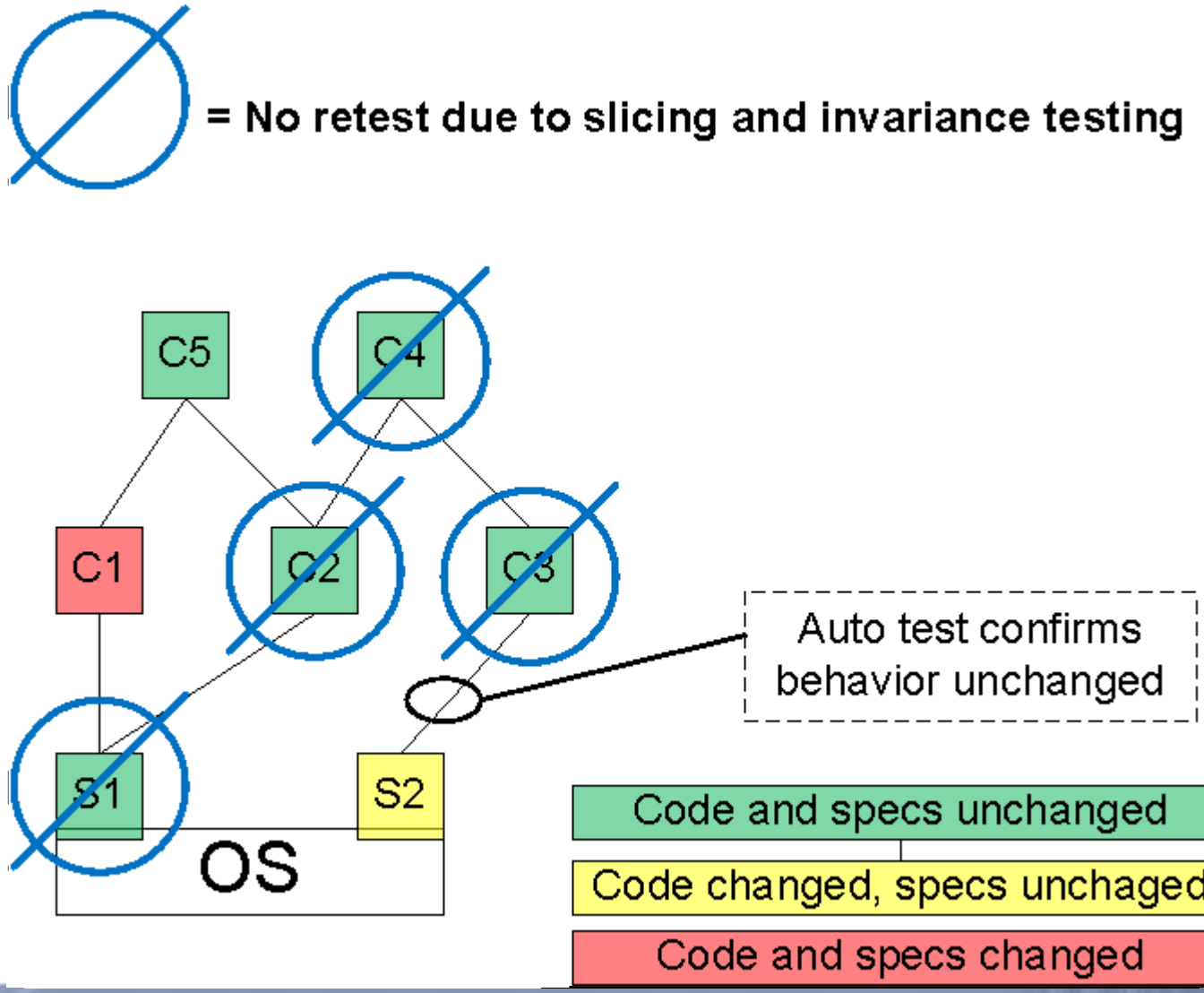
- Maintain safety (this paper)
  - Program slicing to confirm unchanged behavior of unchanged code
  - Automated testing to confirm unchanged behavior of modified code

- Enable Plug-and-Fight (long term vision)
  - Eventually ***eliminate integration test after every reconfiguration***
  - A technology roadmap to accomplish this was presented last year
    - *Proceedings of the Fourth Annual Research Symposium—Acquisition Research: Creating Synergy for Informed Change* (May 16-17 2007, pp. 285-312).
  - This paper addresses a simplified sub-problem of the vision

Retesting Unchanged Components?

- Retesting is necessary *but not always*
- Did component behavior change?
  - Does *it depend on modified code*?
  - Does *the modified code have different behavior*?
- Did component requirements change?
  - Is the *old behavior still appropriate*?
- Did component workload change?
  - Did the *range of valid inputs change*?
  - Did the *range of expected inputs change*?
  - Did the *set of reachable states change*?
- Did available *resources change*?
  - Memory, processor, network bandwidth,…
  - Do other modified components use more resources?

Example


= No retest due to slicing and invariance testing
C5
C4
C1
C2
C3
S1
S2
OS
Auto test confirms behavior unchanged
Code and specs unchanged
Code changed, specs unchaged
Code and specs changed

Approach: Program Slicing [Weiser 84]

- What is a slice?
  - A self-contained subset of a program
    - Contains all of the code that affects its observable behavior
  - Determined by an observation point
    - Example: behavior of a single service
  - Contains only the relevant parts
- Why do slices matter?
  - Behavior invariance property:
    - *If a service has the same slice in two different versions of a program, it has the same behavior in both versions*
  - *If two slices are the same, the service does not have to be retested*
  - Slices can be computed on a large scale
    - Involves dependency tracing, data flow analysis, and control flow analysis

Invariance Testing Extends Program Slicing

- Used to check that behavior of modified code remains the same
  - Candidates: Open Architectures and higher level middleware
    - Enables effective slicing cutoff boundaries
  - Example: operating system interface
  - Example: upgrade from a deprecated interface
  - Example: baseline specific interfaces used by common components
- Enhances slicing to identify more components that do not need retesting
- Relies on a statistical inference with a very high confidence level
  - Needs large numbers of test cases
  - Economically feasible because this kind of test and analysis can be *completely automated*
    - Test cases - generate inputs by random sampling
    - Data analysis - compare outputs from two different software versions

How Much Invariance Testing is Enough?

- How many tests are needed to reach *high confidence*?
  - Stakeholder defines the acceptable risk threshold *k*
    - *The mean time between observations of a behavioral difference in a given operating system service is k-times longer than a mission*.
- Number of test cases is computed for each service in the middleware interface to the operating system
  - It is determined by the following formula

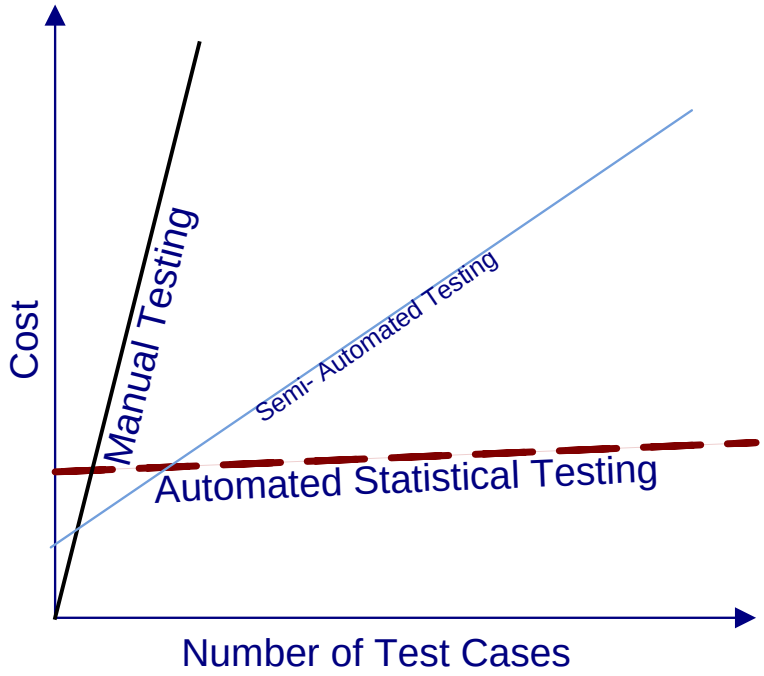
$$T_s = (k\, e_s) \log_2 (k\, e_s)$$

    - Where *s* is a service, e_s is the mean number of executions of *s* per mission, *k* reflects stakeholder's tolerance for risk as above
- Test cases are independently drawn from the probability distribution characterizing the mission, a.k.a. *operational profile*
  - Statistical confidence level is $1 - 1/(k\, e_s)$
    - Probability of making a false positive conclusion matches the stakeholder's risk tolerance

Testing Efforts vs. Acceptable Risk

$N_s = k\ e_s$	C	T_s
10^3	.999	1.0×10^4
10^4	.9999	1.3×10^5
10^5	.99999	1.7×10^6
10^6	.999999	2.0×10^7
10^7	.9999999	2.3×10^8
10^8	.99999999	2.7×10^9
10^9	.999999999	3.0×10^{10}

Number of test cases required for different levels of risk tolerance



Testing cost characteristics

Why Do We Need Operational Profiles

- Can be used to ***automate selection of test cases***

- Reliability of a system is determined by the operational profile
  - Real systems have bugs, coding errors, requirement omission, etc.
  - System reliability varies from **0** (always fails) to **1** (never fails) in different environments

- Operational profiles have proved useful in practice
  - Example: reliability testing of telephone-switching software

- It takes human effort to produce an operational profile
  - Measure the frequency distributions of operating system calls and associated input parameters
    - Can be collected on- or off- line

When Retesting a Service is Necessary

- When its slice or behavior has changed

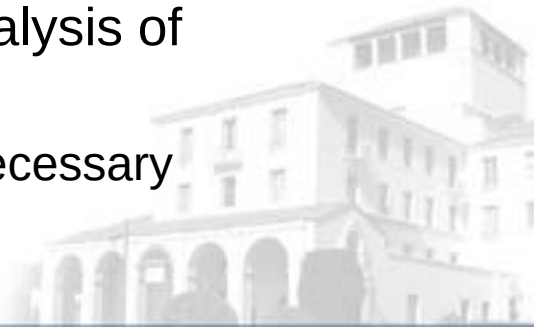
- When requirements have changed
  - New functionality needs to be tested
  - Test all affected components

  - When the ***range of expected operating conditions*** has expanded
  - Even if there was no other change, new test scenarios are needed
  - Indicated by a modified operational profile

- When computing speeds or timing constraints have changed
  - Changed hardware processing rates can adversely affect scheduling algorithms and cause missed deadlines

Conclusions

- The slicing and automated testing approach has a potential to reduce testing duration and costs
  - More research is recommended to substantiate the applicability of our approach to DoD systems
  - Experimental evaluation of slicing method needed
- Automated testing techniques can alleviate concerns about system risks due to technology innovations
- Measurement and analysis of the operational profiles of reusable components can be used to support analysis of changes in the operating environments
  - Hence determining whether additional testing is necessary



Backup Slides

Related Work

- Navy systems are designed with open architecture in mind
  - Hence encapsulating all system calls
- Program Slicing has been used in a wide variety of applications: testing, debugging, program understanding, reverse engineering, software maintenance, change merging, software metrics.
  - See paper for extended list of citations.
- Automate testing has been used to automatically generate open sets of test cases based on random samplings from implementations of operational profile distributions [Berzins and Chaki 2002]
- Prior work on quality assurance for flexible systems at the level:
  - Of requirements [Luqi, Zhang, Berzins & Qiao 2004] [Luqi & Lange 2006]
  - Of architectures [Berzins & Luqi 2006]][Luqi & Zhang 2006]